

SERIAL COMMUNICATION BY USING UART

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND INSTRUMENTATION ENGINEERING

BY

Pradosh Priyadarshan

Roll No.: 10407023

Biswa Ranjan Mundari

Roll No.: 10407026

Under the guidance of:

Prof.K.K.Mahapatra



Department of Electronics and Communication Engineering

National Institute of Technology, Rourkela



CERTIFICATE

This is to certify that the thesis entitled “**Serial Communication using UART**” submitted by **Pradosh Priyadarshan** and **Biswa Ranjan Mundari** in partial fulfillment of the requirements for the award of **Bachelor of Technology** Degree in **Electronics and Instrumentation Engineering** at **National Institute of Technology, Rourkela** (Deemed University) is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Prof K.K.Mahapatra

Date

Department of E.C.E

National Institute of Technology

Rourkela – 769008

ACKNOWLEDGEMENT

We take this opportunity as a privilege to thank all individuals without whose support and guidance we could not have completed our project in this stipulated period of time.

First and foremost we would like to express our deepest gratitude to our Project Supervisor Prof. K.K.Mahapatra, Department of Electronics and Communication Engineering, for his invaluable support, guidance, motivation and encouragement throughout the period this work was carried out.

We are also grateful to Mr. Ayaskanta Swain for their valued suggestions and inputs during the course of the project work. His readiness for consultation at all times, his educative comments and inputs, his concern and assistance even with practical things have been extremely helpful.

We would also like to thank all Professors and Lecturers, and members of the Department of Electronics and Communication for their generous help in various ways for the completion of the thesis. We also extend our thanks to our fellow students for their friendly co-operation.

Pradosh Priyadarshan

10407023

Biswa Ranjan Mundari

10407026

Dept. of E.C.E.

NIT Rourkela

Contents

1 INTRODUCTION

1.1	What is UART and how it works	7
1.2	Serial Vs Parallel	8
1.3	Synchronous Serial Transmission	10
1.4	Asynchronous Serial Transmission	10
1.5	Bits, Baud and Symbols	11
1.6	Asynchronous Serial Reception	13
1.7	Other UART Functions	15

2 RS 232-C STANDARD

2.1	The RS 232-C and V.24 Standards	17
2.2	RS 232-C Bit Assignment (Marks and Spaces)	17
2.3	RS 232-C Break Signal	18
2.4	RS 232-C DTE and DCE Devices	18
2.5	RS 232-C Pin Assignments	19

3 THE 8251 CHIP

3.1	Introduction	23
3.2	Ports	23
3.3	General Operation	27
3.3.1	Programming the 8251	27
3.3.2	The Mode Word	27
3.3.3	Synchronous Mode Word	27
3.3.4	Asynchronous Mode Word	29
3.4	The Command Word and SYNC Characters	32
3.4.1	Command Word	32
3.4.2	The Status Word	33

4 THE 8251 VHDL MODEL

4.1	Introduction	36
4.2	Process Descriptions	36
4.3	Main Process	36
4.3.1	VHDL Code for Main Process	38
4.4	Transmitter Process	40
4.4.1	VHDL Code for Transmitter	42
4.5	Receiver Process	44
4.5.1	Asynchronous Mode	44
4.5.2	VHDL Code for Receiver	46
4.6	VHDL Code for Baud Rate Generator	49

VHDL Program Results

Program Output for Main	52
Program Output for Transmitter	53
Program Output for Receiver	54
Program Output for Clock Divider	56

Conclusion	58
-------------------	----

References	60
-------------------	----

Chapter 1

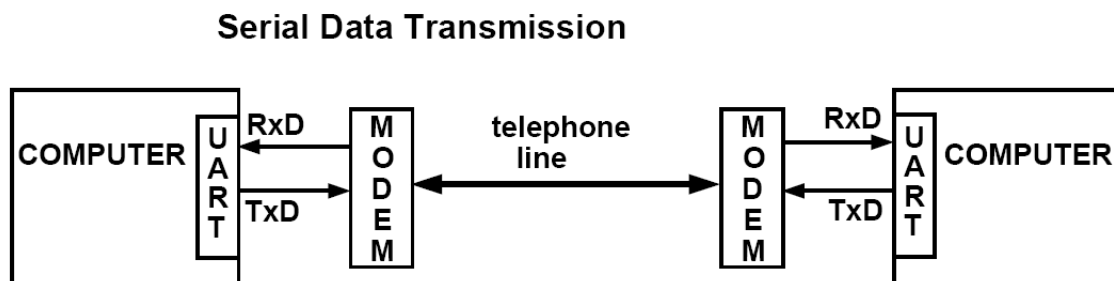
INTRODUCTION

1.1 The UART: What it is and how it works

A UART (Universal Asynchronous Receiver/Transmitter) is the microchip with programming that controls a computer's interface to its attached serial devices. Specifically, it provides the computer with the RS-232C Data Terminal Equipment (DTE) interface so that it can "talk" to and exchange data with modems and other serial devices. As part of this interface, the UART also:

- Converts the bytes it receives from the computer along parallel circuits into a single serial bit stream for outbound transmission
- On inbound transmission, converts the serial bit stream into the bytes that the computer handles
- Adds a parity bit (if it's been selected) on outbound transmissions and checks the parity of incoming bytes (if selected) and discards the parity bit
- Adds start and stop delineators on outbound and strips them from inbound transmissions
- Handles interrupts from the keyboard and mouse (which are serial devices with special ports)
- May handle other kinds of interrupt and device management that require coordinating the computer's speed of operation with device speeds

Serial transmission is commonly used with modems and for non-networked communication between computers, terminals and other devices.



The communications links across which computers—or parts of computers—talk to one another may be either serial or parallel. A parallel link transmits several streams of data

(perhaps representing particular bits of a stream of bytes) along multiple channels (wires, printed circuit tracks, optical fibres, etc.); a serial link transmits a single stream of data.

At first sight it would seem that a serial link must be inferior to a parallel one, because it can transmit less data on each clock tick. However, it is often the case that serial links can be clocked considerably faster than parallel links, and achieve a higher data rate. A number of factors allow serial to be clocked at a greater rate:

- Clock skew between different channels is not an issue (for unclocked serial links)
- A serial connection requires fewer interconnecting cables (e.g. wires/fibres) and hence occupies less space. The extra space allows for better isolation of the channel from its surroundings
- Crosstalk is less of an issue, because there are fewer conductors in proximity.

In many cases, serial is a better option because it is cheaper to implement. Many ICs have serial interfaces, as opposed to parallel ones, so that they have fewer pins and are therefore cheaper.

In telecommunications and computer science, **serial communications** is the process of sending data one bit at one time, sequentially, over a communications channel or computer bus. This is in contrast to parallel communications, where all the bits of each symbol are sent together. Serial communications is used for all long-haul communications and most computer networks, where the cost of cable and synchronization difficulties make parallel communications impractical. Serial computer buses are becoming more common as improved technology enables them to transfer data at higher speeds.

1.2 Serial versus parallel

The Serial Port is harder to interface than the Parallel Port. In most cases, any device you connect to the serial port will need the serial transmission converted back to parallel so that it can be used. This can be done using a UART. On the software side of things, there are many more registers that you have to attend to than on a Standard Parallel Port. (SPP)

So what are the advantages of using serial data transfer rather than parallel?

1. Serial Cables can be longer than Parallel cables. The serial port transmits a '1' as -3 to -25 volts and a '0' as +3 to +25 volts where as a parallel port transmits a '0' as 0v and a '1' as 5v. Therefore the serial port can have a maximum swing of 50V compared to the parallel port which has a maximum swing of 5 Volts. Therefore cable loss is not going to be as much of a problem for serial cables than they are for parallel.
2. You don't need as many wires than parallel transmission. If your device needs to be mounted a far distance away from the computer then 3 core cable (Null Modem Configuration) is going to be a lot cheaper than running 19 or 25 core cable. However you must take into account the cost of the interfacing at each end.
3. Infra Red devices have proven quite popular recently. You may of seen many electronic diaries and palmtop computers which have infra red capabilities build in. However could you imagine transmitting 8 bits of data at the one time across the room and being able to (from the devices point of view) decipher which bits are which? Therefore serial transmission is used where one bit is sent at a time. IrDA-1 (The first infra red specifications) was capable of 115.2k baud and was interfaced into a UART. The pulse length however was cut down to 3/16th of a RS232 bit length to conserve power considering these devices are mainly used on diaries, laptops and palmtops.
4. Microcontrollers have also proven to be quite popular recently. Many of these have in built SCI (Serial Communications Interfaces) which can be used to talk to the outside world. Serial Communication reduces the pin count of these MPU's. Only two pins are commonly used, Transmit Data (TXD) and Receive Data (RXD) compared with at least 8 pins if you use a 8 bit Parallel method (You may also require a Strobe).

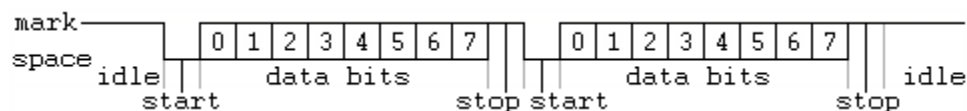
There are two primary forms of serial transmission: Synchronous and Asynchronous. Depending on the modes that are supported by the hardware, the name of the communication sub-system will usually include a **A** if it supports Asynchronous communications, and a **S** if it supports Synchronous communications. Both forms are described below.

1.3 Synchronous Serial Transmission

Synchronous serial transmission requires that the sender and receiver share a clock with one another, or that the sender provide a strobe or other timing signal so that the receiver knows when to “read” the next bit of the data. In most forms of serial Synchronous communication, if there is no data available at a given instant to transmit, a fill character must be sent instead so that data is always being transmitted. Synchronous communication is usually more efficient because only data bits are transmitted between sender and receiver, and synchronous communication can be more costly if extra wiring and circuits are required to share a clock signal between the sender and receiver. A form of Synchronous transmission is used with printers and fixed disk devices in that the data is sent on one set of wires while a clock or strobe is sent on a different wire. Printers and fixed disk devices are not normally serial devices because most fixed disk interface standards send an entire word of data for each clock or strobe signal by using a separate wire for each bit of the word. In the PC industry, these are known as Parallel devices. The standard serial communications hardware in the PC does not support Synchronous operations. This mode is described here for comparison purposes only.

1.4 Asynchronous Serial Transmission

Asynchronous serial communication describes an asynchronous transmission protocol in which a start signal is sent prior to each byte, character or code word and a stop signal is sent after each code word. The start signal serves to prepare the receiving mechanism for the reception and registration of a symbol and the stop signal serves to bring the receiving mechanism to rest in preparation for the reception of the next symbol. A common kind of start-stop transmission is ASCII over RS-232, for example for use in teletypewriter operation.



In the diagram, a start bit is sent, followed by eight data bits, no parity bit and one stop bit, for a 10-bit character frame. The number of data and formatting bits, and the transmission speed, must be pre-agreed by the communicating parties. After the stop bit, the line *may* remain idle indefinitely, or another character may immediately be started.

1.5 Bits, Baud and Symbols

Baud is a measurement of transmission speed in asynchronous communication. Because of advances in modem communication technology, this term is frequently misused when describing the data rates in newer devices. Traditionally, a Baud Rate represents the number of bits that are actually being sent over the media, not the amount of data that is actually moved from one DTE device to the other. The Baud count includes the overhead bits Start, Stop and Parity that are generated by the sending UART and removed by the receiving UART. This means that seven-bit words of data actually take 10 bits to be completely transmitted. Therefore, a modem capable of moving 300 bits per second from one place to another can normally only move 30 7-bit words if Parity is used and one Start and Stop bit are present. If 8-bit data words are used and Parity bits are also used, the data rate falls to 27.27 words per second, because it now takes 11 bits to send the eight-bit words, and the modem still only sends 300 bits per second. The formula for converting bytes per second into a baud rate and vice versa was simple until error-correcting modems came along. These modems receive the serial stream of bits from the UART in the host computer (even when internal modems are used the data is still frequently serialized) and converts the bits back into bytes. These bytes are then combined into packets and sent over the phone line using a Synchronous transmission method. This means that the Stop, Start, and Parity bits added by the UART in the DTE (the computer) were removed by the modem before transmission by the sending modem. When these bytes are received by the remote modem, the remote modem adds Start, Stop and Parity bits to the words, converts them to a serial format and then sends them to the receiving UART in the remote computer, who then strips the Start, Stop and Parity bits. The reason all these extra conversions are done is so that the two modems can perform error correction, which means that the receiving modem is able to ask the sending modem to resend a block of data that was not received with the correct checksum. This checking

is handled by the modems, and the DTE devices are usually unaware that the process is occurring. By stripping the Start, Stop and Parity bits, the additional bits of data that the two modems must share between themselves to perform error-correction are mostly concealed from the effective transmission rate seen by the sending and receiving DTE equipment. For example, if a modem sends ten 7-bit words to another modem without including the Start, Stop and Parity bits, the sending modem will be able to add 30 bits of its own information that the receiving modem can use to do error-correction without impacting the transmission speed of the real data. The use of the term Baud is further confused by modems that perform compression. A single 8-bit word passed over the telephone line might represent a dozen words that were transmitted to the sending modem. The receiving modem will expand the data back to its original content and pass that data to the receiving DTE. Modern modems also include buffers that allow the rate that bits move across the phone line (DCE to DCE) to be a different speed than the speed that the bits move between the DTE and DCE on both ends of the conversation. Normally the speed between the DTE and DCE is higher than the DCE to DCE speed because of the use of compression by the modems. Because the number of bits needed to describe a byte varied during the trip between the two machines plus the differing bits-per-second speeds that are used present on the DTE-DCE and DCE-DCE links, the usage of the term Baud to describe the overall communication speed causes problems and can misrepresent the true transmission speed. So Bits Per Second (bps) is the correct term to use to describe the transmission rate seen at the DCE to DCE interface and Baud or Bits Per Second are acceptable terms to use when a connection is made between two systems with a wired connection, or if a modem is in use that is not performing error-correction or compression. Modern high speed modems (2400, 9600, 14,400, and 19,200bps) in reality still operate at or below 2400 baud, or more accurately, 2400 Symbols per second. High speed modem are able to encode more bits of data into each Symbol using a technique called Constellation Stuffing, which is why the effective bits per second rate of the modem is higher, but the modem continues to operate within the limited audio bandwidth that the telephone system provides. Modems operating at 28,800 and higher speeds have variable Symbol rates, but the technique is the same.

1.6 Asynchronous Serial Reception

A multiplexed data communication pulse can only be in one of two states but there are many names for the two states. When on, circuit closed, low voltage, current flowing, or a logical zero, the pulse is said to be in the "space" condition. When off, circuit open, high voltage, current stopped, or a logical one, the pulse is said to be in the "mark" condition. A character code begins with the data communication circuit in the space condition. If the mark condition appears, a logical one is recorded otherwise a logical zero.

Figure 1 shows this multiplexing format.

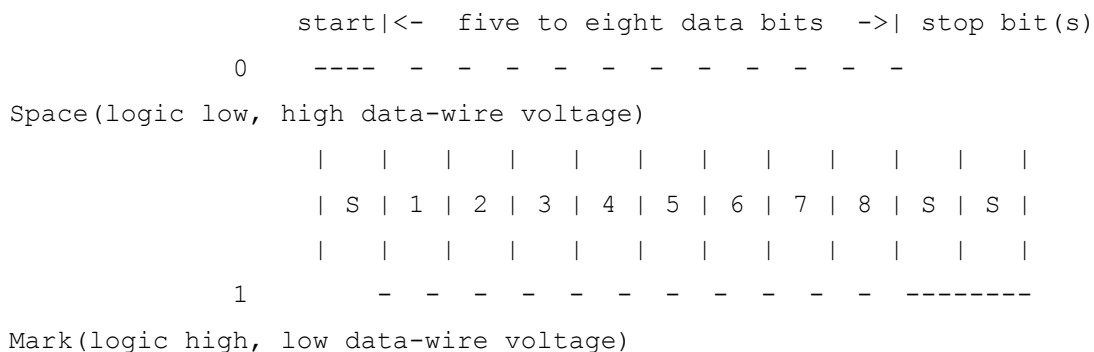


Figure. Asynchronous Code Format.

The least-significant bit is always transmitted first. If parity is present, the parity bit comes after the data bits but before the stop bit(s).

The start bit is always a '0' (logic low), which is also called a **space**. Ironically, the logic low '0' corresponds to a high voltage on the data wire. The start bit signals the receiving DTE that a character code is coming. The next five to eight bits, depending on the code set employed, represent the character. In the ASCII code set the eighth data bit may be a parity bit. The next one or two bits are always in the **mark** (logic high, i.e., '1') condition and called the stop bit(s). They provide a "rest" interval for the receiving DTE so that it may prepare for the next character which may be after the stop bit(s). The rest interval was required by the old mechanical Teletypes which used a motor driven camshaft to

decode each character. At the end of each character the motor needed time to strike the character bail (print the character) and reset the camshaft.

There are six basic steps in receiving a serial character code into a parallel register. First, to keep track of time, the receiver employs a clock which "ticks." When the line is in the space condition, the receiver samples the line 16 times the data rate. In other words, a data interval is equal to 16 clock ticks. In this way the receiver can determine the beginning of the start bit and "move over" to the center of the bit time for data sampling. Second, when the line goes into the mark state, declare a "looking for start bit" condition and wait one half the bit interval or eight clock ticks. Third, sample the line again and if it has not remained in the mark condition, consider this to be a spurious voltage change and go back to step one. Fourth, if the line was still in the mark state, then consider this a valid start bit. Shift the start bit into an eight-bit shift register and wait one bit time or 16 clock ticks. Fifth, after one bit time sample the line (the data should have been there for the last eight clock ticks, and should remain for eight more clock ticks). Now shift the sample into the shift register. Sixth, continue steps four and five seven more times. After the eighth shift, the start bit will "migrate" into a flip-flop indicating character received. Go to step one.

Before the transmitter and receiver UARTs will work, they must also agree on the same values of five parameters. First, both sides must agree on the number of bits per character. Second, the speed or Baud of the line must be the same on both sides. Third, both sides must agree to use or not use parity. Fourth, if parity is used, both sides must agree on using odd or even parity. Fifth, the number of stop bits must be agreed upon. Having said all this, most DTEs today employ eight data bits, no parity, and one stop bit. Thus there is a rule-of-thumb that the number of characters per second is equal to the Baud divided by 10 for a typical RS-232 or RS-423 data line.

1.7 Other UART Functions

In addition to the basic job of converting data from parallel to serial for transmission and from serial to parallel on reception, a UART will usually provide additional circuits for signals that can be used to indicate the state of the transmission media, and to regulate the flow of data in the event that the remote device is not prepared to accept more data. For example, when the device connected to the UART is a modem, the modem may report the presence of a carrier on the phone line while the computer may be able to instruct the modem to reset itself or to not take calls by raising or lowering one more of these extra signals. The function of each of these additional signals is defined in the EIA RS232-C standard.

Chapter 2

THE RS-232 STANDARD

2.1 The RS232-C and V.24 Standards

In most computer systems, the UART is connected to circuitry that generates signals that comply with the EIA RS232-C specification. There is also a CCITT standard named V.24 that mirrors the specifications included in RS232-C.

2.2 RS232-C Bit Assignments (Marks and Spaces)

In RS232-C, a value of 1 is called a Mark and a value of 0 is called a Space. When a communication line is idle, the line is said to be “Marking”, or transmitting continuous 1 values.

The Start bit always has a value of 0 (a Space). The Stop Bit always has a value of 1 (a Mark). This means that there will always be a Mark (1) to Space (0) transition on the line at the start of every word, even when multiple word are transmitted back to back. This guarantees that sender and receiver can resynchronize their clocks regardless of the content of the data bits that are being transmitted.

The idle time between Stop and Start bits does not have to be an exact multiple (including zero) of the bit rate of the communication link, but most UARTs are designed this way for simplicity.

In RS232-C, the "Marking" signal (a 1) is represented by a voltage between -2 VDC and -12 VDC, and a "Spacing" signal (a 0) is represented by a voltage between 0 and +12 VDC. The transmitter is supposed to send +12 VDC or -12 VDC, and the receiver is supposed to allow for some voltage loss in long cables. Some transmitters in low power devices (like portable computers) sometimes use only +5 VDC and -5 VDC, but these values are still acceptable to a RS232-C receiver, provided that the cable lengths are short.

2.3 RS232-C Break Signal

RS232-C also specifies a signal called a `Break`, which is caused by sending continuous Spacing values (no Start or Stop bits). When there is no electricity present on the data circuit, the line is considered to be sending `Break`. The `Break` signal must be of a duration longer than the time it takes to send a complete byte plus Start, Stop and Parity bits. Most UARTs can distinguish between a Framing Error and a `Break`, but if the UART cannot do this, the Framing Error detection can be used to identify `Breaks`. In the days of teleprinters, when numerous printers around the country were wired in series (such as news services), any unit could cause a `Break` by temporarily opening the entire circuit so that no current flowed. This was used to allow a location with urgent news to interrupt some other location that was currently sending information. In modern systems there are two types of `Break` signals. If the `Break` is longer than 1.6 seconds, it is considered a "Modem `Break`", and some modems can be programmed to terminate the conversation and go on-hook or enter the modems' command mode when the modem detects this signal. If the `Break` is smaller than 1.6 seconds, it signifies a Data `Break` and it is up to the remote computer to respond to this signal. Sometimes this form of `Break` is used as an Attention or Interrupt signal and sometimes is accepted as a substitute for the ASCII `CONTROL-C` character. Marks and Spaces are also equivalent to "Holes" and "No Holes" in paper tape systems.

Note: `Breaks` cannot be generated from paper tape or from any other byte value, since bytes are always sent with Start and Stop bit. The UART is usually capable of generating the continuous Spacing signal in response to a special command from the host processor.

2.4 RS232-C DTE and DCE Devices

The RS232-C specification defines two types of equipment: the Data Terminal Equipment (DTE) and the Data Carrier Equipment (DCE). Usually, the DTE device is the terminal (or computer), and the DCE is a modem. Across the phone line at the other end of a conversation, the receiving modem is also a DCE device and the computer that is connected to that modem is a DTE device. The DCE device receives signals on the pins

that the DTE device transmits on, and vice versa. When two devices that are both DTE or both DCE must be connected together without a modem or a similar media translator between them, a NULL modem must be used. The NULL modem electrically re-arranges the cabling so that the transmitter output is connected to the receiver input on the other device, and vice versa. Similar translations are performed on all of the control signals so that each device will see what it thinks are DCE (or DTE) signals from the other device. The number of signals generated by the DTE and DCE devices are not symmetrical. The DTE device generates fewer signals for the DCE device than the DTE device receives from the DCE.

2.5 RS232-C Pin Assignments

The EIA RS232-C specification (and the ITU equivalent, V.24) calls for a twenty-five pin connector (usually a DB25) and defines the purpose of most of the pins in that connector.

In the IBM Personal Computer and similar systems, a subset of RS232-C signals are provided via nine pin connectors (DB9). The signals that are not included on the PC connector deal mainly with synchronous operation, and this transmission mode is not supported by the UART that IBM selected for use in the IBM PC.

Depending on the computer manufacturer, a DB25, a DB9, or both types of connector may be used for RS232-C communications. (The IBM PC also uses a DB25 connector for the parallel printer interface which causes some confusion.)

Below is a table of the RS232-C signal assignments in the DB25 and DB9 connectors.

DB25 RS232-C Pin	DB9 IBM PC Pin	EIA Circuit Symbol	CCITT Circuit Symbol	Common Name	Signal Source	Description
1	-	AA	101	PG/FG	-	Frame/Protective Ground

DB25 RS232-C Pin	DB9 IBM PC Pin	EIA Circuit Symbol	CCITT Circuit Symbol	Common Name	Signal Source	Description
2	3	BA	103	TD	DTE	Transmit Data
3	2	BB	104	RD	DCE	Receive Data
4	7	CA	105	RTS	DTE	Request to Send
5	8	CB	106	CTS	DCE	Clear to Send
6	6	CC	107	DSR	DCE	Data Set Ready
7	5	AV	102	SG/GND	-	Signal Ground
8	1	CF	109	DCD/CD	DCE	Data Carrier Detect
9	-	-	-	-	-	Reserved for Test
10	-	-	-	-	-	Reserved for Test
11	-	-	-	-	-	Reserved for Test
12	-	CI	122	SRLSD	DCE	Sec. Recv. Line Signal Detector
13	-	SCB	121	SCTS	DCE	Secondary Clear to Send
14	-	SBA	118	STD	DTE	Secondary Transmit Data
15	-	DB	114	TSET	DCE	Trans. Sig. Element Timing
16	-	SBB	119	SRD	DCE	Secondary Received Data
17	-	DD	115	RSET	DCE	Receiver Signal Element Timing
18	-	-	141	LOOP	DTE	Local Loopback
19	-	SCA	120	SRS	DTE	Secondary Request to Send

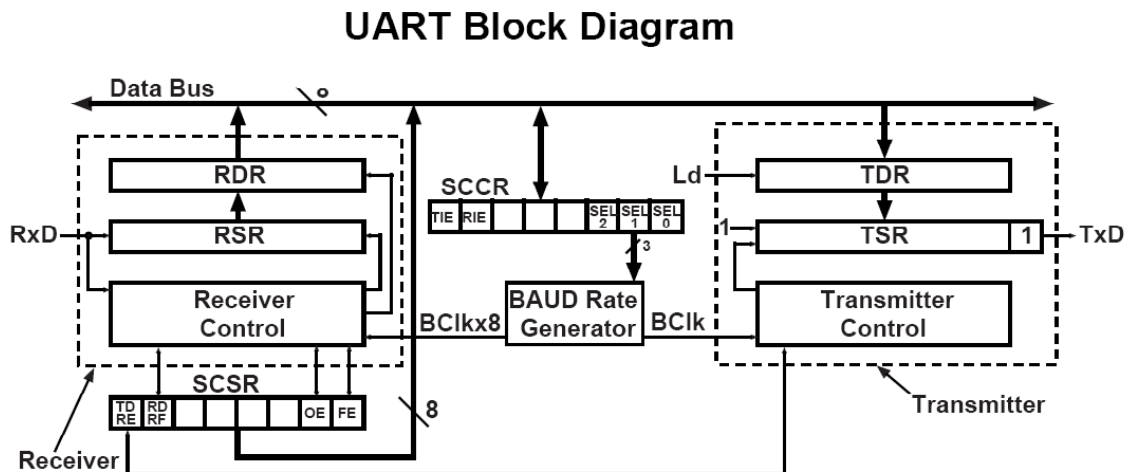
DB25 RS232-C Pin	DB9 IBM PC Pin	EIA Circuit Symbol	CCITT Circuit Symbol	Common Name	Signal Source	Description
20	4	CD	108.2	DTR	DTE	Data Terminal Ready
21	-	-	-	RDL	DTE	Remote Digital Loopback
22	9	CE	125	RI	DCE	Ring Indicator
23	-	CH	111	DSRS	DTE	Data Signal Rate Selector
24	-	DA	113	TSET	DTE	Trans. Sig. Element Timing
25	-	-	142	-	DCE	Test Mode

Chapter 3

THE 8251 CHIP

3.1 Introduction

The Intel 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART), designed for data communication with Intel's microprocessor families. It is used as a peripheral device and is programmed by the CPU to operate using many serial data transmission techniques. The USART accepts data characters from the CPU in parallel format and then converts them into a continuous serial data stream. It accepts serial data streams and converts them into parallel data characters for the CPU. The USART will signal the CPU whenever it can accept a new character for transmission or whenever it has received a character for the CPU. The CPU can read the status of the USART at any time. The status includes data transmission errors and control signals SYNDT/BD, TxEMPTY, TxRDY, RxRDY.



3.2 Ports

RESET (1 BIT, INPUT PORT) :

This is the master reset for the 8251 chip. D₇ to D₀ (8 PINS, 1 BIT each, INOUT PORTS) : These are the bi-directional data bus pins (8 bits) used for transferring

data/control/status words transfer the USART and the CPU. These are usually connected to the CPU's data-bus, although the CPU always remains in control of the bus and initiates all transfers.

CS_BAR (1 BIT, INPUT PORT) :

This is the Chip-Select line. A low on this line enables data communication between the CPU and the USART.

RD_BAR (1 BIT, INPUT PORT) :

This is the read line. A low on this line causes the USART to place the status word or the (received) data word on the data bus pins("D_7" to "D_0").

WR_BAR (1 BIT, INPUT PORT) :

This is the write line. A low on this line causes the USART to accept the data on the data bus pins ("D_7" to "D_0") as either a control word or as a data character (for transmission).

C_D_BAR (1 BIT, INPUT PORT) :

This is the "Control/Data" pin. It is used while transferring data to/from the CPU using the data bus pins ("D_7" to "D_0").

During a read operation : If C_D_BAR - 1, the USART places its status on the data bus pins. If C_D_BAR - 0, the USART places the (received) data character on the data bus pins.

During a write operation : If C_D_BAR - 1, the USART reads a control word from the data bus pins. If C_D_BAR - 0, the USART reads a data character (for transmission) from the data bus pins.

RxD (1 BIT, INPUT PORT) :

This is the receiver data pin. Characters are received serially on this pin and assembled into parallel characters.

TxD (1 BIT, OUTPUT PORT) :

This is the transmitter data pin. Parallel characters received by the CPU are transmitted serially by the USART on this line.

RxC_BAR (1 BIT, INPUT PORT) :

This is the receiver clock. Data on "RxD" is sampled by the USART on the rising edge of "RxC_BAR".

TxC_BAR (1 BIT, INPUT PORT) :

This is the transmitter clock. Data is shifted out serially on "TxD" by the USART, on the falling edge of "TxC_BAR".

CLK (1 BIT, INPUT PORT) :

This clock is used for internal device timing. It needs to be faster than "TxC_BAR" and "RxC_BAR".

TxEMPTY (1 BIT, OUTPUT PORT) :

A high on this line indicates that the serial buffer in the transmitter is empty. This line goes low only while a data character is being transmitted by the USART. It goes high as soon as the USART completes transmitting a character and a new one has not been loaded in time.

TxRDY (1 BIT, OUTPUT PORT) :

This pin signals the CPU that the USART is ready to accept a new data character for transmission. "TxRDY" is reset when the USART receives a data character from the CPU.

RxRDY (1 BIT, OUTPUT PORT) :

This pin signals the CPU that the USART has received a character on its serial input "RxD" and is ready to transfer it to the CPU. "RxRDY" is reset when the character is read by the CPU.

SYNDET_BD (1 BIT, INOUT PORT) :

In the Synchronous mode, this line can be in two ways (while receiving characters). In the "Internal-Synchronization" mode, this line is used as an output which goes high when the programmed "SYNC-characters" are detected on the "RxD" line. In the "External-Synchronization" mode, this line is used as an input and the USART starts assembling data characters at the next clock ("RxC_BAR") edge after a rising edge on this line. In the Asynchronous mode, this line is used as a "Break-Detect" output which goes high if the "RxD" line has stayed low for two consecutive character lengths (including start, stop and parity bits).

RTS_BAR (1 BIT, OUTPUT PORT) :

This "Request-To_Send" is a general purpose output signal that can be asserted by a "command word" from the CPU. It may be used to request that the modem prepare itself to transmit.

CTS_BAR (1 BIT, INPUT PORT) :

This "Clear-To-Send" is an input signal that can be read by the CPU as part of the "status-word". A low on this line enables the USART to transmit data. A low on "CTS_BAR" is normally generated as a response to an assertion on "RTS_BAR".

DTR_BAR (1 BIT, OUTPUT PORT) :

This "Data-Terminal-Ready" is a general purpose output signal that can be asserted by a "command word" from the CPU.

DSR_BAR (1 BIT, INPUT PORT) :

This "Data-Set-Ready" is a general purpose input signal that can be read by the CPU as part of the "status-word".

3.3 General Operation

3.3.1 Programming the 8251

The complete functional definition of the 8251 is programmed by the system's software. A set of control words must be sent out by the CPU to initialize the 8251 to support the desired communication format. These words must immediately follow a reset (internal/external).

3.3.2 The Mode word

Immediately after a reset, the CPU has to send the 8-bit "mode" word. The 8251 can be used for either synchronous/asynchronous data communication. To understand how the mode instruction works, its best to view the device as two separate components, one synchronous and the other asynchronous.

3.3.3 Synchronous mode word

Bit 0 Bit 1
0 0
<div></div> <div></div>

The two least significant bits must be both 0 in Synchronous mode.

Character length : (bits per character)

Bit 3	Bit 2	
-----	-----	-----
0	0	5 bits
_____	_____	_____
0	1	6 bits
_____	_____	_____
1	0	7 bits
_____	_____	_____
1	1	8 bits
_____	_____	_____

Parity :

Bit 5	Bit 4	
-----	-----	-----
0	0	No parity
_____	_____	_____
0	1	Odd parity
_____	_____	_____
1	0	No parity
_____	_____	_____
1	1	Even parity
_____	_____	_____

Bit 4 -- Parity Enable

Bit 5 -- Even Parity

Synchronization scheme :

Bit 7	Bit 6	
0	0	Internal sync detect, Double Sync character
0	1	External sync detect (from SYNDET_BD input)
1	0	Internal sync detect, Single Sync character
1	1	External sync detect (from SYNDET_BD input)

Bit 6 -- External Synchronization

Bit 7 -- Single Sync character (Internal Synchronization)

3.3.4 Asynchronous mode word

Baud Rate : In asynchronous mode, the baud rate defines the number of clock (RxC_BAR/TxC_BAR) cycles over which each bit is transmitted/received. (e.g. At baud rate 64X, each bit is transmitted over 64 clock cycles).

Bit 1	Bit 0	
0	0	Not relevant (Synchronous mode)

0	1	1X baud rate	
_____	_____	_____	
1	0	16X baud rate	
_____	_____	_____	
1	1	64X baud rate	
_____	_____	_____	

Character length : (bits per character)

Bit 3	Bit 2		
-----	-----	-----	
0	0	5 bits	
_____	_____	_____	
0	1	6 bits	
_____	_____	_____	
1	0	7 bits	
_____	_____	_____	
1	1	8 bits	
_____	_____	_____	

Parity :

Bit 5	Bit 4	
0	0	No parity
0	1	Odd parity
1	0	No parity
1	1	Even parity

Bit 4 -- Parity Enable

Bit 5 -- Even Parity

No of Stop Bits :

Bit 7	Bit 6	
0	0	Invalid
0	1	1 stop bit
1	0	1.5 stop bits
1	1	2 stop bits

3.4 The Command word and SYNC characters

In the "Internal Synchronization" mode, the control words (from the CPU) that follow the mode word, must be SYNC characters. In Single-Sync mode, only one SYNC character (SYNC1) is loaded. In Double-Sync mode, two consecutive SYNC characters (SYNC1 followed by SYNC2) must be loaded. The SYNC character(s) have the same number of bits as the data characters (as programmed in the mode word). The SYNC characters (if present, i.e. in "Internal Synchronization" mode) are followed by the command word from the CPU. Data words (for transmission) can follow that. Actually, the command word can be written by the CPU at any time in the data block during the operation of the USART. To write a new Mode word, the master reset in the Command instruction can be set to initiate an "Internal Reset".

3.4.1 Command Word

Bit 0 : Transmitter Enable

Bit 1 : DTR (Data Terminal Ready) -- Controls DTR_BAR output(if this is high, DTR_BAR is low)

Bit 2 : Receiver enable

Bit 3 : Send Break -- Assertion of this forces "TxD" pin low

Bit 4 : Error Reset -- Reset all error flags (parity error, framing error overrun error) in the status word .

Bit 5 : RTS (Request To Send) -- Controls RTS_BAR output (if this is high, RTS_BAR is low)

Bit 6 : Internal Reset -- Resets the USART and makes it ready to accept a new mode word.

Bit 7 : Enter Hunt Mode -- (used only in synchronous receive). If this is high, the USART tries to achieve synchronization by entering the "hunt mode". In "Internal Synchronization" mode, the USART starts looking for the programmed SYNC character(s) at the "RxD" input. In "External Synchronization" mode, the USART starts looking for a rising edge on the "SYNDET_BD" input. Once synchronization is achieved, the USART gets out of "hunt mode" and starts assembling characters at the next rising edge of "RxC_BAR".

3.4.2 The Status Word

The CPU can read the "status word" from the USART at any time.

Bit 0 : TxRDY -- This signifies whether the transmitter is ready to receive a new character for transmission from the CPU. However, in order for the "TxRDY" PIN to be high, three conditions must be satisfied :

- (a) TxRDY STATUS BIT must be high
- (b) "CTS_BAR" must be low
- (c) The transmitter must be enabled (Bit 0 in the Command word must be high).

Bit 1 : RxRDY -- Same as "RxRDY" pin.

Bit 2 : TxEMPTY -- Same as "TxEMPTY" pin.

Bit 3 : Parity Error -- When parity is enabled and a parity error is detected in any received character, this bit is set.

Bit 4 : Overrun Error -- When the CPU does not read a received character before the next one becomes available, this bit is set. However, the previous character is lost.

Bit 5 : Framing Error -- Used only in asynchronous mode. When a valid stop bit (high) is not detected at the end of a received character, this bit is set.

(Note : All three error flags are reset by the "Error Reset" command bit. Also, the setting of these error flags does not inhibit the USART operation.)

Bit 6 : SYNDET_BD -- Same as "SYNDET_BD" pin.

Bit 7 : DSR (Data Set Ready) -- Controlled by "DSR_BAR" pin. (If "DSR_BAR" pin is low, this status bit is high.)

Chapter 4

THE 8251 VHDL MODEL

4.1 Introduction

The VHDL model of the 8251 USART is divided into three major VHDL processes. The processes are called "main", "receiver" and "transmitter". First we will look at the global signals and variables (local to a process) used in the model and their functions. Then, we will briefly study each of the three processes.

4.2 Process Descriptions

The VHDL model consists of three major processes ("main", "receiver" and "transmitter"). We will briefly discuss each of them.

4.3 Main Process

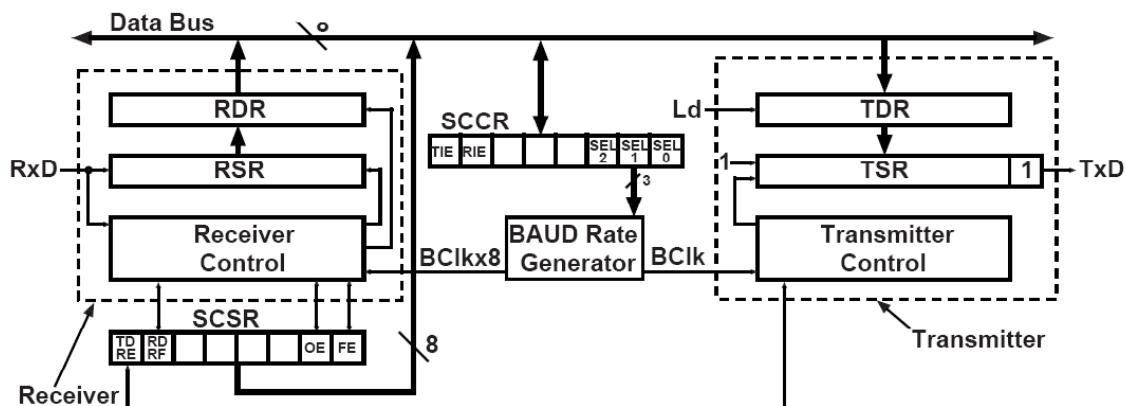
The "main" process has the primary task of being an interface to the CPU. It uses the internal device clock "clk". On getting a reset (internal/external), the main process initializes the global signals, local variables and ports. Then, at every rising clock edge, it checks for signals from the CPU. Since a mode command (from the CPU) must immediately follow a reset, it stores the next CPU control word in the "mode" global signal. On getting the mode word, it computes the following parameters :

- (a) number of bits per character
- (b) number of clock cycles per bit
- (c) number of stop-bit clock cycles (Async mode)
- d) number of clock cycles through which RxD has to remain low for Break-Detect in Asynchronous receive.

These parameters are assigned to global signals and are used by the "transmitter" and "receiver" processes. Then, the process waits for SYNC character(s), if Internal Synchronization Mode has been programmed. The SYNC character(s) are also assigned to global signals for use by the "transmitter" and "receiver" processes. Then the process

waits for a command word from the CPU. (If Internal Synchronization Mode has not been programmed, the process waits for a command immediately after getting a mode word, since no SYNC character(s) are expected.) The command word is assigned to a global signal "command" and various operations (e.g. Internal reset, error flag reset, enter HUNT MODE) are performed, depending on which bits are set in the command word. After this initial phase, other control words sent by the CPU are interpreted as Command words. If the CPU sends data characters for transmission, that character is stored in the global signal "Tx_buffer" (if the transmitter is enabled) and the TxRDY status bit is reset. It also notes whether "CTS_BAR" (clear-to-send input) was low when the character was written, by conditionally setting the global signal "Tx_wr_while_cts". The CPU may want to read a data character that has been received by the USART. In that case, (if the receiver is enabled) the data bits in global signal "Rx_buffer" are placed on the data bus pins and "RxRDY" is reset. The CPU may want to read the status of the USART. In that case, the bits in global signal "status" are placed on the data bus pins and "SYNDET_BD" is reset.

UART Block Diagram



4.3.1 VHDL Code for Main Process:-

```
library ieee;  
use ieee.std_logic_1164.all;  
entity UART is  
port (SCI_sel, R_W, clk, rst_b, RxD : in std_logic;  
ADDR2: in std_logic_vector(1 downto 0);  
DBUS : inout std_logic_vector(7 downto 0);  
SCI_IRQ, TxD : out std_logic);  
end UART;  
architecture uart1 of UART is  
component UART_Receiver  
port (RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;  
RDR: out std_logic_vector(7 downto 0);  
setRDRF, setOE, setFE: out std_logic);  
end component;  
component UART_Transmitter  
port (Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;  
DBUS: in std_logic_vector(7 downto 0);  
setTDRE, TxD: out std_logic);  
end component;  
component clk_divider  
port (Sysclk, rst_b: in std_logic;  
Sel: in std_logic_vector(2 downto 0);  
BclkX8: buffer std_logic;  
Bclk: out std_logic);  
end component;  
signal RDR : std_logic_vector(7 downto 0); -- Receive Data Register  
signal SCSR : std_logic_vector(7 downto 0); -- Status Register
```

```

signal SCCR : std_logic_vector(7 downto 0); -- Control Register
signal TDRE, RDRF, OE, FE, TIE, RIE : std_logic;
signal BaudSel : std_logic_vector(2 downto 0);
signal setTDRE, setRDRF, setOE, setFE, loadTDR, loadSCCR : std_logic;
signal clrRDRF, Bclk, BclkX8, SCI_Read, SCI_Write : std_logic;

begin

  RCVR: UART_Receiver port map(RxD, BclkX8, clk, rst_b, RDRF, RDR, setRDRF,
    setOE, setFE);

  XMIT: UART_Transmitter port map(Bclk, clk, rst_b, TDRE, loadTDR, DBUS,
    setTDRE, TxD);

  CLKDIV: clk_divider port map(clk, rst_b, BaudSel, BclkX8, Bclk);

  -- This process updates the control and status registers
  process (clk, rst_b)
  begin
    if (rst_b = '0') then
      TDRE <= '1'; RDRF <= '0'; OE <= '0'; FE <= '0';
      TIE <= '0'; RIE <= '0';
    elsif (rising_edge(clk)) then
      TDRE <= (setTDRE and not TDRE) or (not loadTDR and TDRE);
      RDRF <= (setRDRF and not RDRF) or (not clrRDRF and RDRF);
      OE <= (setOE and not OE) or (not clrRDRF and OE);
      FE <= (setFE and not FE) or (not clrRDRF and FE);
      if (loadSCCR = '1') then TIE <= DBUS(7); RIE <= DBUS(6);
      BaudSel <= DBUS(2 downto 0);
    end if;
  end if;
  end process;

  -- IRQ generation logic
  SCI_IRQ <= '1' when ((RIE = '1' and (RDRF = '1' or OE = '1'))
    or (TIE = '1' and TDRE = '1'))
  else '0';

```

-- Bus Interface

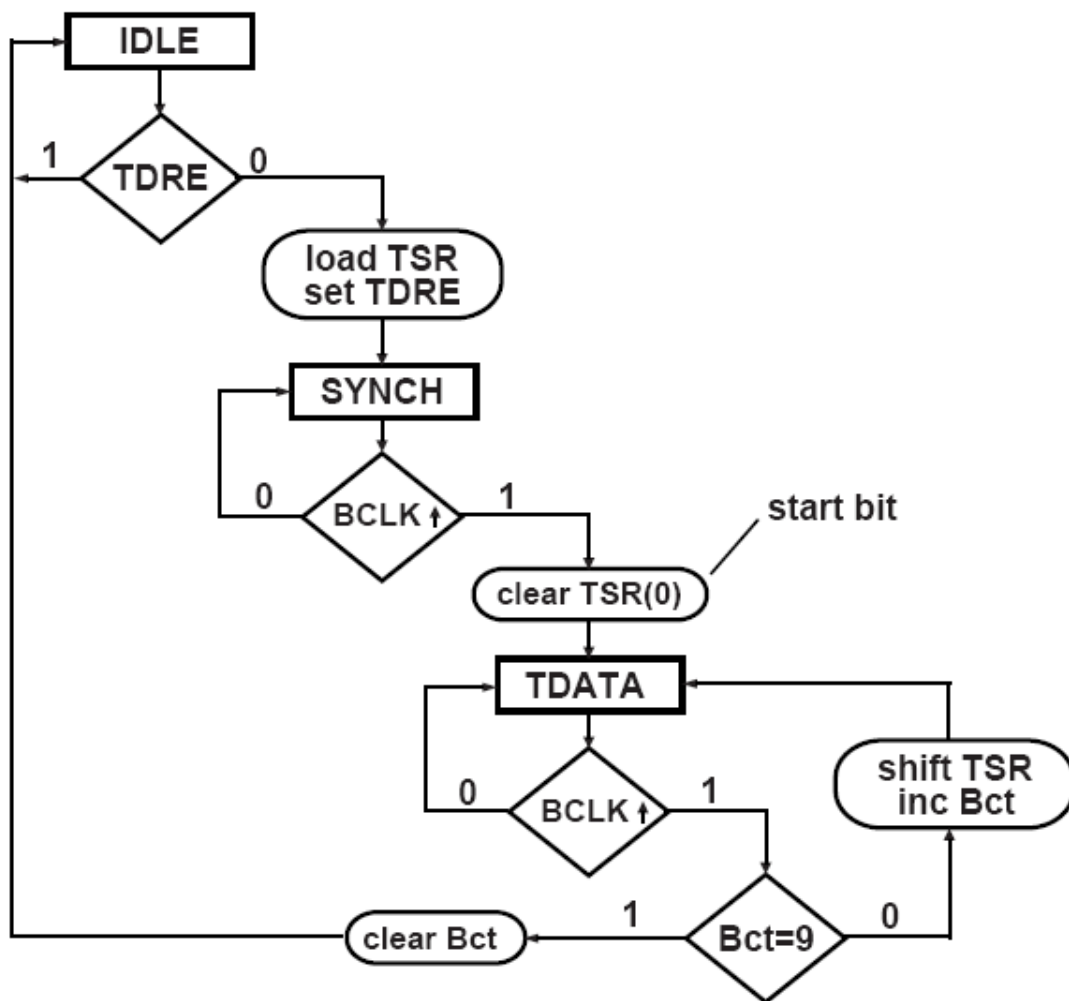
```
SCSR <= TDRE & RDRF & "0000" & OE & FE;
SCCR <= TIE & RIE & "000" & BaudSel;
SCI_Read <= '1' when (SCI_sel = '1' and R_W = '0') else '0';
SCI_Write <= '1' when (SCI_sel = '1' and R_W = '1') else '0';
clrRDRF <= '1' when (SCI_Read = '1' and ADDR2 = "00") else '0';
loadTDR <= '1' when (SCI_Write = '1' and ADDR2 = "00") else '0';
loadSCCR <= '1' when (SCI_Write = '1' and ADDR2 = "10") else '0';
DBUS <= "ZZZZZZZZ" when (SCI_Read = '0') -- tristate bus when not reading
else RDR when (ADDR2 = "00") -- write appropriate register to the bus
else SCSR when (ADDR2 = "01")
else SCCR; -- dbus = sccr, if ADDR2 is "10" or "11"
end uart1;
```

4.4 Transmitter process

The "transmitter" process uses the transmitter clock "TxC_BAR". On getting a reset (internal/external), the transmitter process initializes the global signals, local variables and ports. If the TxRDY status bit is reset (which means that "Tx_buffer" is full), the it checks whether the transmitter has been enabled and "CTS_BAR" (clear-to-send input) is low. If all of these conditions are satisfied or it finds that "Tx_buffer" contains a character that was written while "CTS_BAR" was low (signified by high value on signal "Tx_wr_while_cts"), the transmitter prepares to transmits the data character. If the transmitter is disabled or CTS_BAR is low, he transmitter send a MARKING (high) signal on "TxD" unless it has been commanded to send a BREAK (continous low). In "Internal Synchronization Mode" , if the transmitter is enabled and CTS_BAR is low, but "Tx_buffer" is empty, then the transmitter sends the SYNC character(s) as fillers. For transmitting the data character, the transmitter transfers the data from "Tx_buffer" to "serial_Tx_buffer". It sets the TxRDY status bit because the USART can accept a new character while this one is being transmitted. It also resets TxEMPTY to signify that the "serial_Tx_buffer" is full. If the mode is asynchronous, it first transmits a start bit (low).

In asynchronous mode, every bit is transmitted for a number of clock (TxC_BAR) cycles depending on the baud rate. In synchronous mode, every bit is sent for only one clock cycle. The counter "clk_count" is used to keep track of the number of clock cycles that have passed. Bits are shifted out at the falling edge of TxC_BAR. The data bits are then transmitted. The counter "char_bit_count" is used to count the number of character bits transmitted. If parity is enabled, the appropriate (even/odd) parity bit is transmitted after the data bits. Then, TxEMPTY is reset to signify that the "serial_Tx_buffer" is empty. If the mode is asynchronous, then the required number of stop bits (high) are transmitted.

SM Chart for UART Transmitter



4.4.1 VHDL Code For Transmitter

```
library ieee;
use ieee.std_logic_1164.all;
entity UART_Transmitter is
port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
DBUS:in std_logic_vector(7 downto 0);
setTDRE, TxD: out std_logic);
end UART_Transmitter;
architecture xmit of UART_Transmitter is
type stateType is (IDLE, SYNCH, TDATA);
signal state, nextstate : stateType;
signal TSR : std_logic_vector (8 downto 0); -- Transmit Shift Register
signal TDR : std_logic_vector(7 downto 0); -- Transmit Data Register
signal Bct: integer range 0 to 9; -- counts number of bits sent
signal inc, clr, loadTSR, shftTSR, start: std_logic;
signal Bclk_rising, Bclk_dlayed: std_logic;
begin
TxD <= TSR(0);
setTDRE <= loadTSR;
Bclk_rising <= Bclk and (not Bclk_dlayed); -- indicates the rising edge of bit clock
Xmit_Control: process(state, TDRE, Bct, Bclk_rising)
begin
inc <= '0'; clr <= '0'; loadTSR <= '0'; shftTSR <= '0'; start <= '0';
-- reset control signals
case state is
when IDLE => if (TDRE = '0' ) then
loadTSR <= '1'; nextstate <= SYNCH;
else nextstate <= IDLE; end if;
```

```

when SYNCH => -- synchronize with the bit clock
if (Bclk_rising = '1') then
start <= '1'; nextstate <= TDATA;
else nextstate <= SYNCH; end if;
when TDATA =>
if (Bclk_rising = '0') then nextstate <= TDATA;
elsif (Bct /= 9) then
shftTSR <= '1'; inc <= '1'; nextstate <= TDATA;
else clr <= '1'; nextstate <= IDLE; end if;
end case;
end process;
Xmit_update: process (sysclk, rst_b)
begin
if (rst_b = '0') then
TSR <= "11111111"; state <= IDLE; Bct <= 0; Bclk_dlayed <= '0';
elsif (sysclk'event and sysclk = '1') then
state <= nextstate;
if (clr = '1') then Bct <= 0; elsif (inc = '1') then
Bct <= Bct + 1; end if;
if (loadTDR = '1') then TDR <= DBUS; end if;
if (loadTSR = '1') then TSR <= TDR & '1'; end if;
if (start = '1') then TSR(0) <= '0'; end if;
if (shftTSR = '1') then TSR <= '1' & TSR(8 downto 1); end if; -- shift out one bit
Bclk_dlayed <= Bclk; -- Bclk delayed by 1 sysclk
end if;
end process;
end xmit;

```

4.5 Receiver Process

The "receiver" process uses the transmitter clock "RxC_BAR". On getting a reset (internal/external), the receiver process initializes the global signals, local variables and ports. The receiver process is best understood by considering synchronous and Asynchronous reception separately

4.5.1 Asynchronous mode

If the mode is asynchronous and the receiver is enabled, then RxD is first sampled. If RxD is low, then the receiver is not ready to receive a start bit yet and it waits till the next rising edge on RxC_BAR and samples RxD again. If RxD stays low through a period equal to two character sequences, a break is detected. A counter "brk_count" is used to keep track of the number of clock cycles through which RxC_BAR has stayed low. This counter is reset whenever RxD goes high.

If RxD is high, the receiver is ready to receive a Start-Bit (low). It waits for a falling edge on RxD (Start-Bit). If the baud rate is 16X or 64X, the receiver waits for a number of RxC_BAR cycles equal to half the baud rate (from now, we will call this number "half_baud" -- e.g. for 16X baud rate, "half_baud" will be 8 RxC_BAR cycles). Then it samples RxD again to see if it's still low (False Start-Bit Detection Scheme). If it's still low, then it proceeds to wait for another "half_baud" RxC_BAR cycles and then starts assembling the character. If it's high, this is a false Start-Bit and it goes back to waiting for a falling edge on RxD.

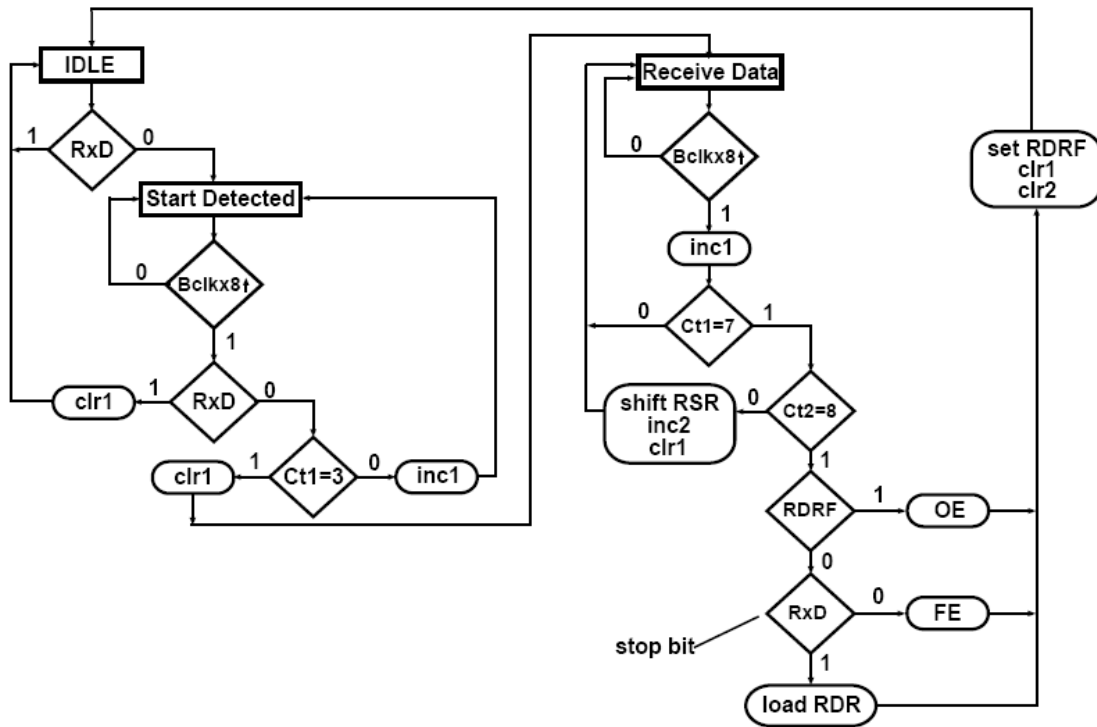
For baud rate 1X, there is no false start-bit detection. For baud rate 1X, data bits and parity bit (if enabled) are sampled at the next rising edge RxC_BAR edge. For baud rate 16X or 64X, data bits and parity bit (if enabled) are sampled at their "nominal center". This is done by waiting for "half_baud" RxC_BAR cycles and then sampling RxD at the

next `RxC_BASR` rising edge. Then, the receiver waits for "half_baud" `RxC_BAR` cycles. The variable "`clk_count`" is used to count the number of `RxC_BAR` cycles passed. The receiver assembles data characters by shifting in bits from `RxD` into the "`serial_Rx_buffer`", for the required number character bits (counted by "`char_bit_count`").

If parity has been programmed, then the parity bit is received after the data bits and parity error is checked. The parity error flag is set, if error is detected. The assembled is transferred from the "`serial_Rx_buffer`" to the "`Rx_buffer`". (The CPU can read this buffer, via the "main" process.) If the receiver is enabled, `RxRDY` is set to signal the CPU that a received character is waiting to be read.

If the previously received character (if any) is still unread by the CPU, then this new character overwrites it and overrun error is detected. Then, the receiver checks the stop bit (high) at the next rising edge of clock. If `RxD` is low, then framing error flag is asserted.

SM Chart for UART Receiver



4.5.2 VHDL code for Receiver

```

library ieee;
use ieee.std_logic_1164.all;
entity UART_Receiver is
port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
RDR: out std_logic_vector(7 downto 0);
setRDRF, setOE, setFE: out std_logic);
end UART_Receiver;
architecture rcvr of UART_Receiver is
type stateType is (IDLE, START_DETECTED, RECV_DATA);
signal state, nextstate: stateType;
signal RSR: std_logic_vector (7 downto 0); -- receive shift register
signal ct1 : integer range 0 to 7; -- indicates when to read the RxD input
signal ct2 : integer range 0 to 8; -- counts number of bits read
signal inc1, inc2, clr1, clr2, shftRSR, loadRDR : std_logic;

```

```

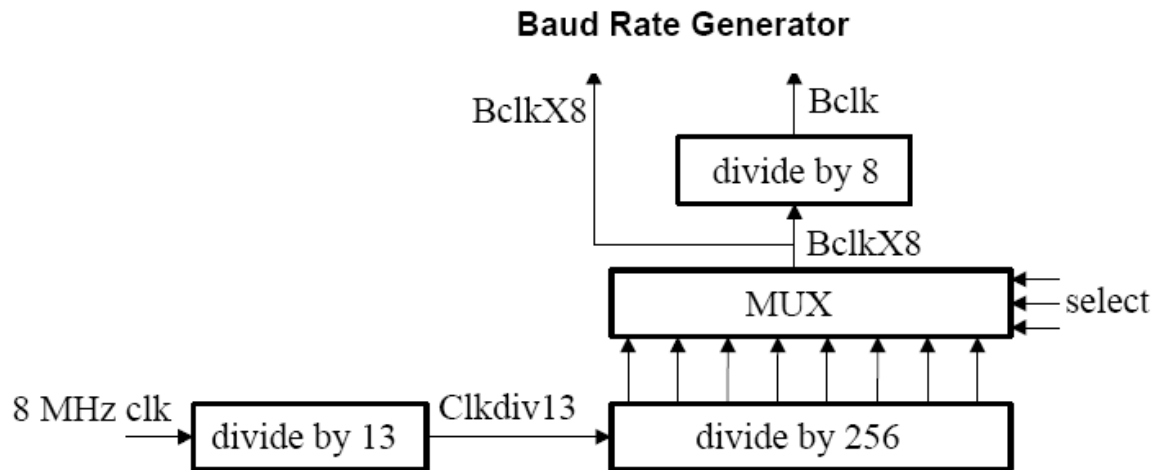
signal BclkX8_Dlayed, BclkX8_rising : std_logic;
begin
BclkX8_rising <= BclkX8 and (not BclkX8_Dlayed);
-- indicates the rising edge of bitX8 clock
Rcvr_Control: process(state, RxD, RDRF, ct1, ct2, BclkX8_rising)
begin
-- reset control signals
inc1 <= '0'; inc2 <= '0'; clr1 <= '0'; clr2 <= '0';
shftRSR <= '0'; loadRDR <= '0'; setRDRF <= '0'; setOE <= '0'; setFE <= '0';
case state is
when IDLE => if (RxD = '0' ) then nextstate <= START_DETECTED;
else nextstate <= IDLE; end if;
when START_DETECTED =>
if (BclkX8_rising = '0') then nextstate <= START_DETECTED;
elsif (RxD = '1') then clr1 <= '1'; nextstate <= IDLE;
elsif (ct1 = 3) then clr1 <= '1'; nextstate <= RECV_DATA;
else inc1 <= '1'; nextstate <= START_DETECTED; end if;
when RECV_DATA =>
if (BclkX8_rising = '0') then nextstate <= RECV_DATA;
else inc1 <= '1';
if (ct1 /= 7) then nextstate <= RECV_DATA;
-- wait for 8 clock cycles
elsif (ct2 /= 8) then
shftRSR <= '1'; inc2 <= '1'; clr1 <= '1'; -- read next data bit
nextstate <= RECV_DATA;
else
nextstate <= IDLE;
setRDRF <= '1'; clr1 <= '1'; clr2 <= '1';
if (RDRF = '1') then setOE <= '1'; -- overrun error
elsif (RxD = '0') then setFE <= '1'; -- framing error
else loadRDR <= '1'; end if; -- load recv data register

```

```

end if;
end if;
end case;
end process;
Rcvr_update: process (sysclk, rst_b)
begin
if (rst_b = '0') then state <= IDLE; BclkX8_Dlayed <= '0';
ct1 <= 0; ct2 <= 0;
elsif (sysclk'event and sysclk = '1') then
state <= nextstate;
if (clr1 = '1') then ct1 <= 0; elsif (inc1 = '1') then
ct1 <= ct1 + 1; end if;
if (clr2 = '1') then ct2 <= 0; elsif (inc2 = '1') then
ct2 <= ct2 + 1; end if;
if (shftRSR = '1') then RSR <= RxD & RSR(7 downto 1); end if;
-- update shift reg.
if (loadRDR = '1') then RDR <= RSR; end if;
BclkX8_Dlayed <= BclkX8; -- BclkX8 delayed by 1 sysclk
end if;
end process;
end rcvr;

```

Select Bits	BAUD Rate (<i>Bclk</i>)
000	38462
001	19231
010	9615
011	4808
100	2404
101	1202
110	601
111	300.5

4.6 VHDL Code for Baud Rate Generator

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; -- use '+' operator, CONV_INTEGER func.
entity clk_divider is
port(Sysclk, rst_b: in std_logic;
Sel: in std_logic_vector(2 downto 0);
BclkX8: buffer std_logic;
Bclk: out std_logic);
end clk_divider;
architecture baudgen of clk_divider is
signal ctr1: std_logic_vector (3 downto 0):= "0000"; -- divide by 13 counter
signal ctr2: std_logic_vector (7 downto 0):= "00000000"; -- div by 256 ctr

```

```

signal ctr3: std_logic_vector (2 downto 0):= "000"; -- divide by 8 counter
signal Clkdiv13: std_logic;
begin
process (Sysclk) -- first divide system clock by 13
begin
if (Sysclk'event and Sysclk = '1') then
if (ctr1 = "1100") then ctr1 <= "0000";
else ctr1 <= ctr1 + 1; end if;
end if;
end process;
Clkdiv13 <= ctr1(3); -- divide Sysclk by 13
process (Clkdiv13) -- clk_divdr is an 8-bit counter
begin
if (rising_edge(Clkdiv13)) then
ctr2 <= ctr2 + 1;
end if;
end process;
BclkX8 <= ctr2(CONV_INTEGER(sel)); -- select baud rate
process (BclkX8)
begin
if (rising_edge(BclkX8)) then
ctr3 <= ctr3 + 1;
end if;
end process;
Bclk <= ctr3(2); -- Bclk is BclkX8 divided by 8
end baudgen;

```

VHDL PROGRAM

RESULTS

All the VHDL programs have been implemented and tested on Xilinx Project Navigator Release ISE 8.2i. The detailed summary of outputs are given below.

PROGRAM OUTPUT FOR MAIN:-

Design Name	UART
Fitting Status	Successful
Software Version	I.31
Device Used	XA2C128-7-VQ100
Date	11-23-2007, 12:35PM

RESOURCES SUMMARY

Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
78/128 (61%)	168/448 (38%)	74/128 (58%)	17/80 (22%)	131/320 (41%)

PIN RESOURCES

Signal Type	Required	Mapped	Pin Type	Used	Total
Input	5	5	I/O	12	70
Output	2	2	GCK/IO	1	3
Bidirectional	8	8	GTS/IO	3	4
GCK	1	1	GSR/IO	1	1

GTS	0	0	CDR/IO	0	1
GSR	1	1	DGE/IO	0	1

GLOBAL RESOURCES

Signal mapped onto global clock net (GCK0)	clk
Signal mapped onto global output enable net (GSR)	rst_b

PROGRAM OUTPUT FOR TRANSMITTER:-

Design Name	UART_Transmitter
Fitting Status	Successful
Software Version	I.31
Device Used	XA2C32A-6-VQ44
Date	11-23-2007, 12:36PM

RESOURCES SUMMARY

Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
26/32 (82%)	54/112 (49%)	24/32 (75%)	15/33 (46%)	37/80 (47%)

PIN RESOURCES

Signal Type	Required	Mapped	Pin Type	Used	Total
Input	11	11	I	0	1
Output	2	2	I/O	9	24
Bidirectional	0	0	GCK/IO	1	3
GCK	1	1	GTS/IO	4	4
GTS	0	0	GSR/IO	1	1
GSR	1	1			

GLOBAL RESOURCES

Signal mapped onto global clock net (GCK0)	sysclk
Signal mapped onto global output enable net (GSR)	rst_b

PROGRAM OUTPUT FOR RECEIVER:-

Design Name	UART_Receiver
Fitting Status	Successful
Software Version	I.31

Device Used	XA2C32A-6-VQ44
Date	11-23-2007, 12:39PM

RESOURCES SUMMARY

Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
30/32 (94%)	58/112 (52%)	26/32 (82%)	16/33 (49%)	45/80 (57%)

PIN RESOURCES

Signal Type	Required	Mapped	Pin Type	Used	Total
Input	3	3	I	0	1
Output	11	11	I/O	10	24
Bidirectional	0	0	GCK/IO	1	3
GCK	1	1	GTS/IO	4	4
GTS	0	0	GSR/IO	1	1
GSR	1	1			

GLOBAL RESOURCES

Signal mapped onto global clock net (GCK0)	sysclk
Signal mapped onto global output enable net (GSR)	rst_b

PROGRAM OUTPUT FOR CLOCK DIVIDER:-

Design Name	clk_divider
Fitting Status	Successful
Software Version	I.31
Device Used	XA2C32A-6-VQ44
Date	11-23-2007, 12:41PM

RESOURCES SUMMARY

Macrocells Used	Pterms Used	Registers Used	Pins Used	Function Block Inputs Used
16/32 (50%)	25/112 (23%)	15/32 (47%)	6/33 (19%)	18/80 (23%)

PIN RESOURCES

Signal Type	Required	Mapped	Pin Type	Used	Total
Input	3	3	I	0	1
Output	2	2	I/O	3	24
Bidirectional	0	0	GCK/IO	1	3
GCK	1	1	GTS/IO	2	4
GTS	0	0	GSR/IO	0	1
GSR	0	0			

GLOBAL RESOURCES

Signal mapped onto global clock net (GCK0)	Sysclk
---	--------

Conclusion

Hence, in the project titled “ **Serial Communication using UART** ” the functioning of a 8251 chip according to RS232 – C standards was studied with VHDL programming.

Pradosh Priyadarshan

Roll : 10407023

Biswa Ranjan Mundari

Roll : 10407026

References:

1. www.wikipedia.org
2. www.google.com
3. VHDL Primer by J. Bhasker
4. The 8088 and 8086 Microprocessors by Walter A. Triebel & Avtar Singh

